VŠB - TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2013 **Pavel Balcárek**

VŠB - TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
KATEDRA INFORMATIKY

Vývoj subsystémů herního enginu pro RTS

Development of Game Engine Subsystems for RTS

2013 **Pavel Balcárek**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Pavel Balcárek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vývoj subsystémů herního enginu pro RTS
Development of Game Engine Subsystems for RTS

Zásady pro vypracování:

Cílem práce je vytvoření několika subsystémů herního enginu pro hry typu FPS. Při realizaci zadání se snažte volit co nejefektivnější postupy a algoritmy. Implementaci proveďte v jazyce C++ a dodržujte Google C++ Style Guide. Grafický subsystém bude založen na OpenGL. Zvažte také možnost využití aplikačních akcelerátorů typu NVIDIA PhysX pro vytvoření fyzikálně korektního prostředí. V textu práce popište zejména algoritmy a metody, které jste použil při řešení zadaných oblastí.

1. Seznamte se s typickými architekturami herních enginů.
2. Navrhněte následující subsystémy: import objektů z vhodného modelovacího nástroje včetně animací, měření vzdáleností v prostředí (geodetická vzdálenost na trojúhelníkové síti), hledání nejkratší cesty, jednoduchá AI.
4. Výsledky průběžně kombinujte s paralelně vyvíjenými moduly.
5. Funkčnost celého systému ověřte na jednoduchém demu, pro které vytvořte vhodné modely.

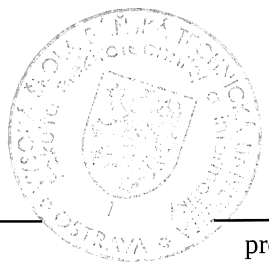
Seznam doporučené odborné literatury:

[1] Gregory, J. Game Engine Architecture. 2009. ISBN 9781568814131.
[2] McShaffry, M. Game Coding Complete. Third edition. 2009. ISBN 1584506806.
[3] Ericson, C. Real-Time Collision Detection. 2005. ISBN 9781558607323.
[4] Millington, I., Funge, J. Artificial Intelligence for Games. Second edition. 2009. ISBN 9780123747310.
[5] Wright, R. S., Lipchak, B., Haemel, N. S. OpenGL(R) SuperBible: Comprehensive Tutorial and Reference. Fifth edition. 2010. ISBN 0321712617.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 18.11.2011
Datum odevzdání: 07.05.2013



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6.5.2013 Podpis: 

Rád bych zde poděkoval všem, kteří mi s touto prací pomohli. Panu Ing. Tomáši Fabiánovi za vedení, podnětné nápady a připomínky, dále celé rodině za podporu a toleranci mého časového vytížení a Martinu Dorazilovi za spolupráci na výsledném demu.

Abstrakt

Práce se zabývá seznámením se s architekturou herního enginu pro realtime strategii a řešením problémů, které tento celek obsahuje nebo může obsahovat. Hlavním zaměřením je logický subsystém realtime strategie. Pod tento obecný celek spadá zejména problém hledání nejkratší cesty v terénu a jednoduchá AI jednotek. Pro tyto problémy bude představena implementace za použití známých algoritmů a postupů a pro prezentaci bude vytvořena demo aplikace za použití open source technologií.

Klíčová slova: *Bakalářská práce, Strategická hra, Hledání nejkratší cesty v terénu, Jednoduchá herní AI, Rozhodovací stromy, Binární halda, A* algoritmus, Dijkstrův algoritmus, Konečné stavové automaty*

Abstract

This thesis deals with architecture of realtime strategy engine and solving problems which can appear while developing realtime strategy game. Main focus is on logical subsystem, mainly finding shortest path in terrain and simple AI for autonomnous units. Demo application will be developed, while containing implementations of all problems theoreticaly described before with using known algorithms and techniques.

Keywords: *Bachelor thesis, Strategy game, Pathfinding, Simple game AI, A-star algorithm, Dijkstras algorithm, Binary heaps, Decision trees, Finite state machines*

Seznam použitých zkratek

RTS	Real-time Strategy
FPS	First person shooter
AI	Artificial intelligence
A*	A Star algorithm
FSM	Finite state machine
VBO	Vertex buffer object
SDL	Simple Directmedia Layer
UML	Unified Modelling Language

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Počítačové hry urazily od svého vzniku dlouhou cestu. Je tomu již více než půl století co se objevily první náznaky toho, co v dnešní době za počítačové hry opravdu považujeme. Jelikož jsem tyto počítačové hry hrál již od mala, vždy mě fascinovalo co způsobuje to, že se daná věc pohybuje a provádí to co jsem jí přikázal.

V roce 1992 se na trhu objevila hra Dune 2 a popularizovala subžánr strategické hry probíhající v reálném čase. Realtime Strategie, jak je tento subžánr nazýván, od té doby prošla dlouhým vývojem ve všech oblastech. Zaměřením této práce je seznámit se s architekturou RTS her, jak lze navrhnout jednotlivé subsystémy a vytvořit z nich hru jako celek. Hlavním zaměřením je logický subsystém RTS her. Zde se nalézá velké množství problému, které je nutno pochopit a navrhnout jejich implementaci tak, abychom byli s výsledkem spokojeni.

V první kapitole se zabývám RTS hrami jako celkem, jejich vznikem a historií. Také obecně popisuji jednotlivé subsystémy a jejich použití v konkrétních situacích.

V druhé kapitole se zabývám hledáním nejkratší/nejoptimálnější cesty v terénu za použití grafových algoritmů a jejich implementaci, převedením 3D terénu do zpracovatelné podoby a následným exportem výsledku tak, aby jej byly schopny zpracovat ostatní objekty. Také musím zmínit použití různých druhů pro výpočet odhadované vzdálenosti, která může velice ovlivnit výslednou cestu.

V třetí kapitole se zabývám problematikou implementace jednoduché AI pro jednotky do logického subsystému. Jelikož tato problematika je velice obsáhlá, popisuji zde spíše konkrétní příklady užití různých technik ve specifických případech. Úkolem je popsat jednoduché struktury, které můžeme použít k definování chování jednotlivých jednotek. Dále řeším chování jednotky při samotném pohybu. Aby jednotka byla schopna pohybu, je zde několik problémů které musím vyřešit, aby výsledný pohyb byl korektní a nepůsobil na hráče špatným dojmem. Zmíním se i o dynamickém obnovování cesty a jeho vlivu na výslednou cestu.

V poslední kapitole je popsán jednoduchý příklad implementace importování 3D modelů do enginu hry a způsobu jejich animace.

Celkovým cílem této práce je vhodně implementovat tyto specifické oblasti do enginu hry jako samostatný modul, který se stará o logiku hry. Poté tento modul zkombinovat s paralelně vyvíjeným grafickým subsystémem, a vytvořit tak funkční engine pro real-time strategii, která by eventuálně mohla být dále vyvíjena a rozšiřována.

2 Realtime strategie

Strategická hra je hra založená na přemýšlení a plánování dalších kroků k dosáhnutí vítězství. Hráč musí přemýšlet a rozhodovat o následujících akcích, které mají ve výsledku vést k dosažení cíle. Cíle se různí v závislosti na typu strategické hry. V dnešní době často rozlišujeme dva hlavní typy počítačových strategických her. Jedním jsou tahové strategie, tedy takové kdy v jednom okamžiku hraje jen jeden hráč a druhý čeká na svůj tah. Druhým typem je realtime strategie. Realtime strategie je založena na tom, že veškeré akce probíhají v reálném čase a hráči tedy nečekají na akci svého protihráče.

Za typickou tahovou strategii s cílem eliminovat soupeře se dají považovat šachy. Je to příklad tahové strategie, tedy takové, kdy se hráči střídají ve svých akcích. Hráč musí dopředu přemýšlet a plánovat své kroky k tomu aby dosáhl cíle, jímž je poražení soupeře. Na podobném principu vzniklo mnoho počítačových her, které přinesly obrovské množství různých vylepšení, možných akcí a různých cílů. Jmenovitě například série Heroes of Might and Magic. Je to tahová strategie, kde každý hráč buduje své město, případně více měst a ovládá své hrdiny. Hra je rozdělena na dvě části, z nichž obě se odehrávají na tahy. První částí je pohyb po mapě, kde hráč ovládá svého hrdinu, který má předem určený počet kroků, které v jednotlivých tazích může vykonat. Prozkoumává tak svět, objevuje zdroje potřebné pro provoz svého města a potkává nepřátele. Druhou částí je souboj, kde proti sobě stojí dva hrdinové se svými vojáky. Střídají se v akci se svými jednotkami a cílem není nic jiného, než porazit nepřítelovu armádu. Každá jednotka má specifické schopnosti a promyšlenými tahy lze porazit nepřítele i s menší armádou než má on.

Realtime strategie se oproti tahové liší hlavně v způsobu hraní. Na rozdíl od tahové strategie hráč nečeká na akce ostatních hráčů, ale všichni provádí své akce v reálném čase paralelně. To urychluje celkový proces hry jako takové, ale zároveň to klade na hráče požadavky na ne jen přemýšlení o dalších akcích, ale i reakcích na situace které vzniknou v průběhu. Celý tento žánr jak je známe dnes prakticky definovala hra Dune 2 od společnosti Westwood Studios. Základní principy této hry, i když mnohokrát velice upravené, se používají dodnes. Hráč si mohl vybrat jednu ze tří ras, za kterou hrál. Průběh hry spočíval jak v budování města a těžení zdrojů, tak v budování armády a soubojů. Hráč ovládá své jednotky tak, že jim dává příkazy a ty jej ihned vykonávají. Může je tak libovolně přesunovat, rozkázat k zaútočení či ústupu. Další hrou, která prakticky stanovila standart od kterého se dále vývoj realtime strategií odvíjel byla hra Age of Empires od Ensemble Studios. Tato hra byla postavena na enginu Genie, který byl později použit i pro pokračování této hry. Engine této hry obsahuje subsystemy funkčností podobné těm, které se dnes používají. Jmenovitě to jsou grafický subsystém založeny na 2D izometrickém pohledu hráče na herní mapu, zvukový subsystém přehrávající zvuky jednotek, prostředí a hudbu. Dále také síťový subsystém, který slouží ke hře více hráčů a logický subsystém který určuje chování jednotek. Age of Empires také používá pro pathfinding A* algoritmus, ale je známo, že implementace není nijak dokonalá. Existují bugy, které například zabraňují jednotce se pohnout, pokud je součástí skupiny více jednotek.

Dnešní moderní realtime strategie také sestávají z několika subsystému, které jako celek mají za úkol poskytnout hráči jak herní tak vizuální zážitek. Výsledkem spolupráce těchto subsystému je celek který se nazývá engine. Na rozdíl od FPS se na poli realtime strategií enginy nelicencují v takovém množství. Většinou vývoji hry předchází vývoj enginu, který je pak v různých modifikovaných verzích použit pro další hry.

Příkladem může být engine Bang! frimy Ensemble Studios. Tento engine byl vydán 25. Října 2002 a nejprve použit pro hru Age of Mythology. Později byl modifikován a použit pro velice úspěšný titul Age of Empires III.

2.1 Subsystémy RTS game enginu

- Grafický subsystém
- Zvukový subsystém
- Vstupní subsystém
- Herní subsystém
  - Logika hry(Pravidla, cíle)
  - AI logika jednotek
    - Hledání cesty
    - Rozhodovací systém jednotek
    - Management jednotek
- Sítový subsystém

Prakticky všechny moderní realtime strategie v dnešní době používají grafický subsystém založený buď to na Direct3D nebo OpenGL. Obojí jsou to API, které mohou být použity pro vykreslování 2D a 3D scény. Direct3D je proprietární API a lze jej použít jen na platformách společnosti Microsoft. Naproti tomu OpenGL je multiplatformní. Lze jej používat na většině dnes známých operačních systémech(i mobilní zařízení)

V našem enginu jsme pro vývoj jednotlivých subsystémů použili SDL. SDL neboli Simple Direct-Media Layer je multiplatformní knihovna, která poskytuje přístup k audio, video a vstupním zařízením. Usnadňuje tudíž přístup ke vstupním zařízením a umožňuje zpracovávání vstupních událostí. Má vlastní systém pro zpracování událostí, které uchovává ve frontě. Typicky potom průběh herní smyčky vypadá následovně:

1. Zpracování vstupu
2. Zpracování událostí a logika hry
3. Vykreslení snímku na obrazovku

Dále nám SDL poskytuje funkce pro práci s vícevláknovým zpracováním událostí, čehož lze na dnešním hardwaru výhodně využít pro paralelní zpracování jednotlivých subsystémů. Táké nám umožňuje velice jednoduše implementovat zvukový subsystém a případně i síťový. Všechny tyto subsystémy tvoří dohromady celek, který se nazývá herní engine.

Engine je tedy množina komponent, které vývojářům slouží k dalšímu vývoji hry samotné. Například tedy zobrazovací subsystém, načítání modelů, zpracování vstupních zařízení, logický subsystém, hledání cesty, zvukový subsystém a mnoho dalších. Naproti tomu samotná data(modely, zvuky, atd.), pravidla, příběh a podobné věci už tvoří hru samotnou.

Jak můžeme vidět v novodobých herních titulech, této skutečnosti se velice hojně využívá, protože ne vždy je výhodné vyvíjet svůj vlastní engine. Často se tedy enginy licencují různým společnostem, které poté postaví vlastní hru na těchto licencovaných enginech.

3 Problém průchodnosti terénu a nalezení optimální cesty

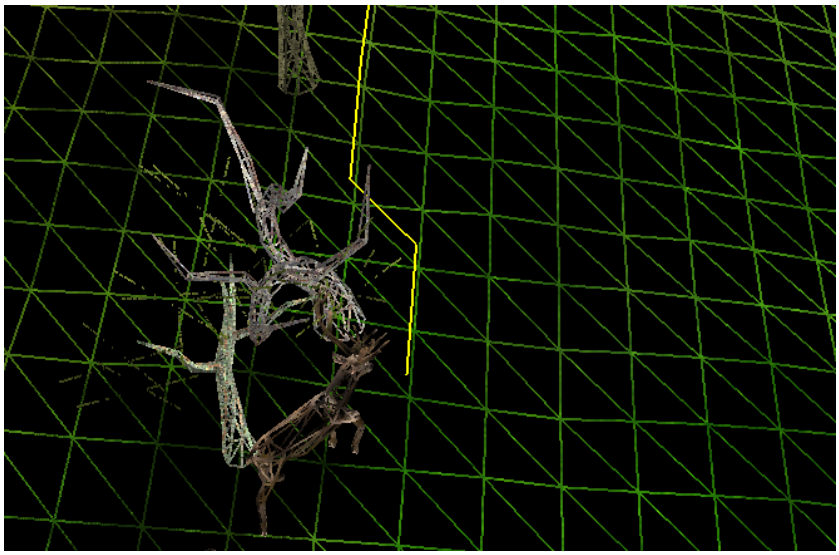
Jeden z nejdůležitějších subsystému logické části RTS enginu je systém pro hledání cesty nebo-li pathfinder. Jednotky pohybující se v prostoru potřebují znát cestu k cíli, která může vést i přes členitý terén a různé překážky. Jednotce může být v kteroukoliv chvíli přikázáno aby se vydala do nějakého cílového bodu. Autonomní jednotky, které hráč neovládá, také potřebují znát cestu k cíli, aby mohly vykonávat další úkoly. Pro všechny tyto případy musí být subsystém pro hledání cesty být schopen jednotce v rozumném čase poskytnout validní cestu, pokud taková existuje.

V této kapitole se věnuji jak teorii ohledně hledání cesty a její implementace, tak také problémům které se při implementaci objevily. Jelikož hledání cesty je kromě celého grafického subsystému jedním z nejnáročnějších výpočetních úkolů, při samotné implementaci musí být kladeny požadavky na použití správných algoritmů a programátorských postupů. Důležitá je také optimalizace pro konkrétní použití.

3.1 Úvod do problematiky

Je zřejmé, že od počátku k cíli nemusí vést pouze jedna cesta. Přirozeně chceme, aby cesta kterou se jednotka vydá, byla pokud možno co nejkratší, nebo nejrychlejší. Výslednou cestu může ovlivnit velké množství faktorů, např. stromy, voda, jiné pohybující se jednotky. Dále je nutné brát v potaz typ terénu po kterém se jednotka pohybuje, jelikož různé terény se mohou lišit svojí náročností. Stejně jako je pro nás přirozené jít po chodníku místo po trávě a kamenech, tak lépe na hráče působí jednotka, která upřednostňuje stejný výběr cesty. Právě toho lze dosáhnout různým ohodnocením náročnosti cesty.

Jelikož většina algoritmů použitelných pro hledání nejkratší cesty pracuje s nezáporně ohodnocenými grafy, tak je důležité, abychom herní mapu reprezentovali ve zpracovatelné podobě. Prakticky to znamená promítnout 3D terén do 2D podoby a ten použít pro zpracování. Po zpracování následuje zpětné převedení celé cesty zpět do datové podoby, kterou je jednotka schopna zpracovat a cestu vykonat.



Obrázek 1: Ukázka reprezentace cesty - RTS Engine

3.2 Algoritmy použitelné pro hledání

Algoritmů, které řeší výše uvedený problém je více. Mezi nejznámější patří Dijkstrův algoritmus. Poprvé ho popsal Edsger Dijkstra v roce 1959. Tento algoritmus se dá použít pro nalezení nejkratší cesty v kladně ohodnoceném grafu. Je konečný, protože v každém průchodu se do množiny uzlů, které již byly navštíveny přidá právě jeden uzel. Průchodů cyklem, muže být maximálně tolik, kolik máme v grafu vrcholů. Problémem tohoto algoritmu je, že hledá cestu do každého možného vrcholu, což je zbytečné pro hledání cesty ze startovního bodu do bodu cílového. V tomto případě nám daleko lépe poslouží následující algoritmus, který se nazývá A-Star(zkráceně také A*).

3.3 A Star algoritmus

Většina RTS her, které vznikly v posledních patnácti letech používá pro systém hledání cesty A* algoritmus [4]. Poprvé jej roku 1968 popsali P.Hart, N.Nilsson a B.Raphael. Algoritmus má tu výhodu, že stejně jako BFS používá heuristiku(odhadovanou vzdálenost k cíli) a zároveň zohledňuje cenu cesty od počátku stejně jako Dijkstra. To znamená, že na rozdíl od Dijkstrova algoritmu v každé iteraci hledání zpracovává bod, který pravděpodobně povede k cíli po nejkratší cestě. Další výhodou tohoto algoritmu je to, že je relativně jednoduchý na implementaci a snadno modifikovatelný. A* algoritmus používá dva listy bodů. Open-list je list dostupných vrcholů k prozkoumání, closed-list je seznam již prozkoumaných vrcholů. Vrchol obsahuje dané informace nutné ke zpracování:

- g - Aktuální cena cesty od počátku
- h - Odhadovaná vzdálenost od cíle
- f - Hodnotu můžeme vyjádřit tímto vzorcem: $f(x) = h(x) + g(x)$

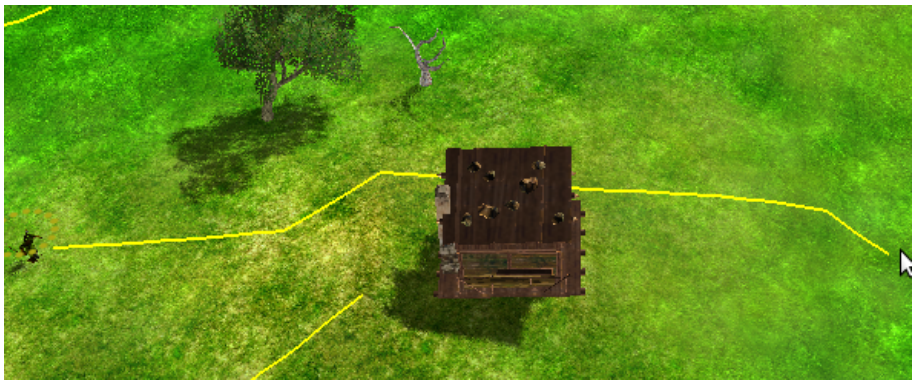
Zjednodušeně tedy algoritmus funguje následovně:

1. Vezmeme počáteční bod A a vložíme jej do open-listu.
2. Opakujeme následující
    - Vybereme bod A z nejmenší hodnotou f.
    - Prozkoumáme všechny body v okolí a pokud jsou průchozí(tj. neobsahují žádnou překážku, nebo nejsou už obsazené) přidáme je také do open-listu. Všem bodům nastavíme jako předka bod A. To je důležité pro zpětnou rekonstrukci cesty.
    - Přesuneme bod A do closed-listu, protože už jej nebudeme nyní potřebovat.
    - Vybereme bod z nejmenší hodnotou f z open-listu.
    - Pokud je A cílový bod, ukončíme vyhledávání. Jinak pokračujeme.
3. Zpětně rekonstruujeme cestu

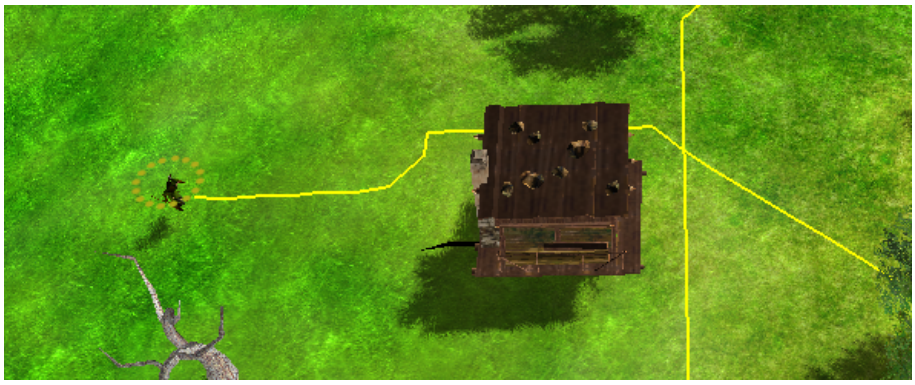
Zpětná rekonstrukce probíhá tak, že pokud v hledacím cyklu dojdeme do cílového bodu hledání se zastaví. Každý bod si uchovává informace o svém předkovi, tedy o bodu z kterého jsme se do aktuálního dostali. Zpětně tedy projdeme body a uložíme každý z nich. Tímto dostaneme kompletní cestu, kterou pak může entita zpracovat.

Důležitou roli v chování tohoto algoritmu představuje odhadování vzdálenosti. Heuristická funkce h(n) nám poskytne informaci o pravděpodobné vzdálenosti z konkrétního bodu do cíle. Při použití různého způsobu výpočtu mohou nastat následující situace:

- Pokud se odhadovaná h(n) rovná nule, A* algoritmus se změní v Dijkstrův a prohledá všechny body než narazí na cílový. V tomto případě vždy dostaneme nejkratší cestu, za cenu velice pomalého zpracování.
- Pokud je h(n) vždy menší než cena pohybu z bodu do bodu, A* nalezne vždy nejkratší cestu, ale prozkoumá více bodů, než je potřeba.
- Pokud je h(n) stejná jako cena pohybu z bodu n do cíle, A* bude následovat jen nejlepší cestu. Toho nelze dosáhnout ve všech případech, existují takové případy, kdy lze této skutečnosti využít.
- Pokud je h(n) větší než cena z konkrétního bodu do cíle, není zaručeno, že nalezená cesta bude nejkratší. Na druhou stranu hledání cesty proběhne rychleji.
- Pokud je h(n) mnohokrát větší vzhledem k ceně cesty od počátku, bude záležet převážně na odhadované vzdálenosti a A* se tak bude chovat stejně jako BFS algoritmus.



Obrázek 2: Příklad vyvážené heuristiky - obcházení je po větším oblouku



Obrázek 3: Příklad přeceněné heuristiky - všimněte si rozdílu oproti předchozímu příkladu

Díky této skutečnosti můžeme algoritmus modifikovat tak, jak se nám zlíbí a co je pro subsystém přínosem. Je celkem zřejmé, že výsledná cesta nemusí být vždy ta nejkratší. Výhodnější je spíše taková která se k nejkratší co nejvíce blíží a zároveň nezpomaluje algoritmus. To znamená najít vyváženy poměr mezi rychlostí výpočtu a délkou výsledné cesty. Na mřížce která se používá pro A* algoritmus jsou popsány tyto metody pro spočítání odhadované vzdálenosti h(n).

- Manhattan distance - Pojmenováno podle uspořádání cest v ulicích Manhattanu. Tato metoda se nejčastěji používá pro A* na mřížce, kde je možný pohyb jen na 4 strany.

```
dx = abs(node.x - goal.x)
dy = abs(node.y - goal.y)
return D * (dx + dy)
```

- Chebyshew distance - Někdy také nazývána šachová vzdálenost podle počtu skoků, které musí král vykonat než dosáhne daného místa. Používá se na mřížce, kde je možný i diagonální pohyb.

```
dx = abs(node.x - goal.x)
dy = abs(node.y - goal.y)
return D * max(dx, dy)
```

- Euclidean distance - Používá se v případě, že pohyb je možný do všech směrů pod libovolným úhlem.

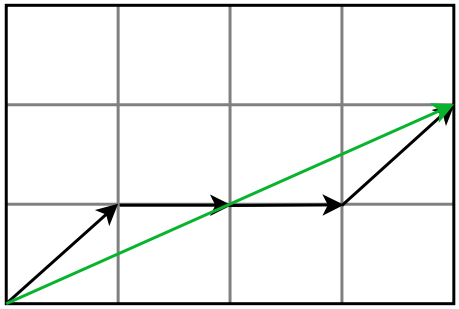
```
dx = abs(node.x - goal.x)
dy = abs(node.y - goal.y)
return D * sqrt(dx * dx + dy * dy)
```

3.4 Možné vylepšení algoritmu pro hledání cesty

Použití A* algoritmu s sebou nese výhody poměrně jednoduché modifikace celého subsystému pro hledání cesty. Profesionální použití tohoto algoritmu v komerčních hrách a aplikacích počítá například s víceúrovňovým hledáním cesty. To znamená, že mapa jako celek je rozdělená do jednotlivých oblastí. Tyto oblasti jsou pak dále rozděleny na podoblasti. Pathfinder tudíž na delší vzdálenost využije hrubší síť průchodnosti a až se nalézá v poslední cílové oblasti, potom použije jemnější síť pro nalezení cesty do cílového bodu. Toto vylepšení má významný vliv na rychlost celého algoritmu, neboť zmenší počet vrcholů pro vyhledávání řádově 10 až 100 krát.

Při vhodném navržení celého subsystému můžeme také zohlednit stoupání nebo klesání cesty. Jedním ze způsobů, jak toho docílit je přidat jednotlivým bodům atribut výška. Potom při prozkoumávání okolních bodů porovnáme také výšku okolních bodů a stoupání či klesání můžeme zohlednit ve výsledné ceně cesty. Cena výsledné cesty je jedním z atributů, které se dají velice lehce modifikovat a přináší výhodu věrného chování jednotek při pohybu. Například můžeme ohodnotit okolí budov nižší cenou cesty než cenu cesty okolo stromů. Při použití dvou různých vrstev tak můžeme dosáhnout toho, že městské jednotky jako například vesničané budou vyhledávat cestu spíše přes město a naopak zvěř bude volit zalesněnou oblast a městu se vyhýbat. Díky této vlastnosti můžeme tedy opravdu libovolně modifikovat celý způsob hledání cesty tak, aby spíše než nejkratší cestu hledal cestu nejpravděpodobnější.

Dalším vylepšením je vyhlazování cest. Jelikož na mřížce A* algoritmus produkuje cesty, kde jednotka se pohybuje přesně po jednotlivých vrcholech. Výsledkem je sekaný pohyb [5]. Pokud chceme dosáhnout reálnějšího pohybu jednotky, je nutno použít právě vyhlazování cest. Dobrým příklad můžete vidět na obrázku dole. Vyhlazování cesty tímto způsobem je docela jednoduché, dokud uvažujeme stejnou cenu pohybu z bodu do bodu. Způsob spočívá v postupném odebírání bodů, dokud máme cíl v dohledu po přímce. Je to velice jednoduchý způsob, ale musíme brát ohled na implementaci kolizí. Pokud kolize řešíme na úrovní jednotlivých bodů, tak vyvstane problém s jednotkami, které by v jistých případech mohly projít skrz sebe. Tento způsob by se tedy dal výhodně použít v případě použití více systému pro detekci kolize.



Obrázek 4: Ukázka vyhlazování cesty

3.5 Implementace hlednání v 2D síti

V našem RTS enginu je hledání cesty implementováno následovně. Třída pathfinder, která má jen jednu instanci pro celý engine, se stará o veškeré požadavky na nalezení cesty. S postupem času byly implementovány i metody pro kolizní systém jednotek, zamlouvání pozic a hledání nejbližších volných pozic v síti. Struktura Node reprezentuje vrchol grafu.

Zdrojový kód 1: Struktura Node

```
struct Node
{
  int walkability;
  int terrainCost;
  void *object;
};
```

Při vytvoření instance pathfinderu se inicializuje dvourozměrné pole typu struktury Node. Po vygenerování mapy v jádru enginu, se volá funkce updateMapData, která zajišťuje korektní načtení mapy do tohoto dvourozměrného pole. Kontroluje se výška terénu, podle které se určuje hladina vody a tímto získáváme 2D pole s charakteristikou terénu. Jak můžeme vidět na ukázce zdrojového kódu 2, je tento způsob velice snadno modifikovatelný co se týče přidání různých vlastností terénu(výška, typ povrchu a s ním spojená cena cesty).

Zdrojový kód 2: jednoduché vytvoření 2D profilu mapy

```
void PathFinder::updateMapData(float waterLevel)
{
  int size = terrain->get_size();
  for (int x = 0; x < size; x++)
  {
    for (int y = 0; y < size; y++)
    {
      //offset by one.
      if (terrain->get_value(x, y) > waterLevel)
      {
        node[x + 1][y + 1].walkability = WALKABLE;
        node[x + 1][y + 1].terrainCost = 1;
      }
      else
      {
        node[x + 1][y + 1].walkability = UNWALKABLE;
```

```
            node[x + 1][y + 1].terrainCost = 5;
        }
    }
  }
}
```

Zdrojový kód 3: Metody pro rezervování pozice

```
int ReservePosition(int x, int y, void* entityObject);
int ReservePosition(int x, int y);
bool ReserveBuildPosition(DataTypes::Cube *cPlace, glm::vec2 pos, void* obj);
```

Jelikož to by k samotnému hledání cesty nestačilo, uvažujeme-li další entity, nacházející se na mapě, existují v pathfinderu metody reservePosition. Tyto metody slouží k zamluvení pozice jakékoliv entity na mapě. Prakticky tak veškeré jednotky si při svém vytvoření nejprve zjistí, zda-li je možné obsadit tuto pozici. Pokud ano, walkability je nastavena na neprůchozí hodnotu a do proměnné objekt je uložen ukazatel na entitu okupující tuto pozici.

Mít uložen odkaz na jednotku přímo v této struktuře s sebou nese značné výhody. Značně to usnadňuje vyhledávání jednotek v okolí, nehledě na možnost interakce s okolím. Více o tomto problému a způsobu implementace lze nalézt v kapitole 5.6

Proměnná walkability určuje průchodnost terénu, terrainCost je cena průchodu a object je ukazatel na objekt, který může danou node okupovat. Walkability by se dalo reprezentovat jako proměnná typu boolean, nicméně v současném stavu může nabývat více hodnot. To slouží k virtuální průchodnosti vody. Ta se řeší až na logické úrovni jednotky při zamlouvání pozice. Voda je tak sice pro pathfinder průchozí, ale nastavení ceny terénu na vysokou hodnotu zajišťuje, že výsledná cesta nikdy nevede přes vodu. To zajišťuje jak věrné chováni jednotky, tak minimalizaci nároků na výpočetní výkon.

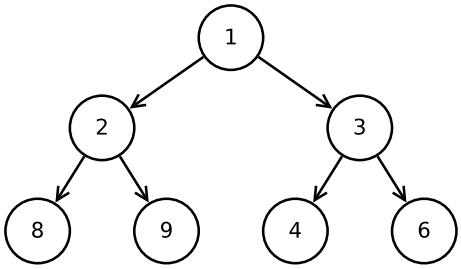
Mějme dva ostrovy které jsou odděleny vodou a pošleme jednotku cíleně z jednoho ostrova na druhý. V případě že by daná situace nebyla ošetřena tak jako nyní, pathfinder by musel prohledat veškeré vrcholy, než by došel k závěru že cesta neexistuje. Nyní pathfinder najde nejkratší cestu mezi ostrovy a předá ji jednotce. Jednotka sama musí rozpoznat zda li je pro ni voda průchozí či ne. Toto chování významně přispívá k omezení nároků na pathfinder a celkově tak na plynulost hry samotné.

Samotná implementace pathfinderu pomocí A* algoritmu není na první pohled složitá. Problém se později objeví v použitých datových strukturách pro open-list. Jelikož A* algoritmus v každé iteraci vybírá z open-listu prvek s nejmenší hodnotou f, je důležité si ujasnit, jak postupovat. Pokud bychom uvažovali velice jednoduchý příklad s malým počtem vrcholů(tj. malá mapa, nebo velice hrubá síť), můžeme uvážit použití vyhledávání prvku s nejmenší hodnotou f v každé iteraci. Jak jsem ale dále zjistil, tento způsob je díky rychlosti prakticky nepoužitelný pro velké mapy a větší počet jednotek. Proto musíme zvolit takovou strukturu, která nám umožní velice rychlý přístup k prvku s nejmenší hodnotou. Pro tento případ nám nejlépe poslouží struktura, která se nazývá binární halda.

3.5.1 Implementace open-listu pomocí binární haldy

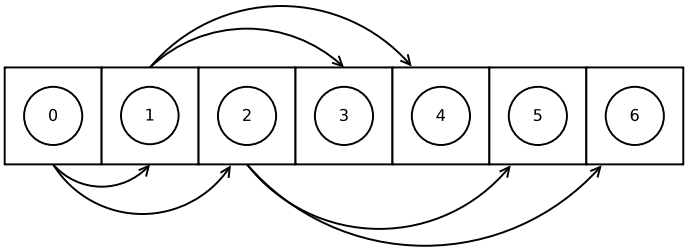
Jedním z pro nás nejlepších způsobů jak si uchovávat data v open-listu je držet tento list setříděný tak, abychom vždy nejmenší prvek měli jako první. Tím zaručíme okamžitý přístup k prvku s nejmenší hodnotou f. Jednou z možností by bylo provést po každém vložení nového prvku do open-listu jeho setřídění. Tato operace by ale byla časově zbytečně náročná, protože nepotřebujeme celý seznam vrcholů v open-listu mít setříděný, ale stačí aby jsme si byli jisti tím, že první prvek je zároveň ten s nejmenší hodnotou f. K tomu nám výborně poslouží následující struktura.

Binární halda je úplný binární strom, který podle typu haldy slouží k rychlému hledání minima či maxima, podle typu stromu [6]. Lze jej velice jednoduše reprezentovat v klasickém poli [7].



Obrázek 5: Binární halda

To jak je binární halda reprezentovaná v poli si můžeme nejlépe představit na tomto obrázku.



Obrázek 6: Reprezentace binární haldy v poli

Časové náročnosti jednotlivých operací jsou následující.

- Přístup k prvku s nejmenší hodnotou - $O(1)$
- Vložení prvku - $O(\log n)$
- Odebrání minimálního prvku a rekonstrukce validní binární haldy $O(\log n)$

Open-list je tedy v mé implementaci reprezentován jako jednorozměrné pole. Velikost toho pole je dynamická a momentálně, jelikož mapa je čtverec, nabývá druhé mocniny počtu bodů na jedné straně. Obsahuje identifikační číslo prvku, podle kterého se pak zjišťují další hodnoty bodu, hlavně pak hodnota f, která je na příslušném ID v poli Fcost a podle kterého se halda řadí. Tedy veškeré datové struktury pro pathfinder jsou reprezentovány jako jedno nebo dvourozměrné pole.

Zdrojový kód 4: Deklarace proměnných pro pathfinder

```
int *openList;  // Jednorozměrné pole, držící identifikátor bodu.
int *openX;  // Pole držící x hodnotu bodu záznamu v˜open-listu
int *openY; // Pole držící y hodnotu bodu záznamu v˜open-listu
int *Hcost; // Pole pro uchování h hodnoty bodu v˜open-listu
int *Fcost; // Pole uchovávající f hodnotu bodu v˜open-listu
int **parentX; // Pole pro uchování x souřadnice předka bodu
int **parentY; // Pole pro uchování y souřadnice předka bodu
int **Gcost; // Pole pro uchování g hodnoty bodu
```

```
int **whichList; // Dvourozměrné pole, uchovávající záznam zda-li se bod nachází na
    open či closed listu
```

Vstupním bodem pro vyhledání validní cesty je metoda pathfinderu getPath. Ta zajistí inicializaci samotného hledaní cesty(uvolnění zdrojů z předchozích hledání) a také se postará o korektní cíl. Pokud jako vstupní argument přijde nedostupné místo na mapě, metoda se postará o nalezení nejbližšího volného bodu. Tato funkčnost je zvlášť výhodná uvážíme-li např. centrální řízení jednotky, která od budovy dostane příkaz na návrat domů a předá ji jako cíl cesty svou pozici. Přirozeně tato pozice je již obsazená a tak jednotka musí nalézt nejbližší bod, na který okolo budovy může směřovat.

V prvním pokusu implementace tato funkce nebyla dostupná a k vyhledání cesty nedošlo, pokud nebyl zadán cíl jako validní průchozí bod. Tato skutečnost ale později přinesla problémy ve spojitosti s autonomním chováním jednotky a tak byla dodatečně implementována. Velice to usnadnilo pozdější implementaci AI subsystému pro jednotlivé jednotky.

Po nalezení validního bodu na mapě se spouští samotné vyhledání cesty pomocí A* algoritmu. Pokud hledání proběhne úspěšně, následuje zpětná rekonstrukce cesty. Ta probíhá od posledního bodu, který je zároveň bodem cílovým. Samotný průběh je následující. Vezme se cílový bod, uloží se do struktury pathBank. Zjistí se předek tohoto bodu, tedy bod ze kterého jsme se dostali do stávajícího a uloží se taky. Takto zpětně rekonstruujeme celou cestu.

Posledním procesem, který následuje je převedení této cesty do datového typu zpracovatelného jednotkou. Pro sjednocení datových typů pro logický subsystém je použit datový typ glm::vec2 z knihovny GL Math, která obsahuje datové typy zpracovatelné grafickým subsystémem. Díky tomu je dosaženo jednotného typů pro zpracování pozic a cest v logickém subsystému.

4 Umělá inteligence v RTS

4.1 Úvod do game AI

Pokud si položíme otázku co je AI, zjistíme, že je velice těžké na ni odpovědět. Tento pojem je velice široký a dodnes není jasné co vše do něj zapadá a co ne. V podstatě je to schopnost programu provádět rozhodnutí a akce, kterých jsou v reálném světě schopni lidé a zvířata [1][2]. Přesto ale tento pojem může být zavádějící. Pokud zavedeme pojem Game AI, tak zjistíme, že ve výsledku až tak nezáleží na tom co se děje uvnitř celého tohoto procesu, ale jak se celá AI chová navenek. To prakticky znamená že vhodnou kombinací různých technik a postupů se snažíme dosáhnout chování, které se pro hráče bude jevit jako inteligentní.

První hry obvykle neobsahovaly žádnou AI, nebo velice primitivní. Jako příklad můžeme uvést hru Space Invaders. Cílem hry bylo sestřelit nepřátele, kteří se po obrazovce dolu přibližovali k hráči. V tomto případě se zde nenalézá žádná logika, jen předem určený pohyb nepřítele směrem dolů, který se s postupem na vyšší úroveň zrychloval.

Zajímavým příkladem jednoduché, ale přesto zábavné AI protivníku představuje hra Pac-Man. Podstata hry je velice jednoduchá. Máte Pac-Mana, který musí v každé úrovni posbírat všechny kuličky v bludišti. V tom se mu snaží zabránit 4 na první pohled stejní nepřátele - duchové, kteří se liší jen barvou. Při dalším zkoumání ale zjistíme, že se liší i způsobem, jakým hráče pronásledují [3]. Například červený duch vždy pronásleduje hráče přímo na jeho pozici, růžový však 4 pole před Pac-manův aktuální směr atd. Kombinací těchto různých algoritmů pro pronásledování mohou vznikat různé situace a hra tudíž nepůsobí vždy stejně. Podle mého názoru je tohle krásný příklad toho, jak s různorodým, ale přesto jednoduchým chováním nepřítele, dosáhnout vysoké hratelnosti.

Pokud se poohlédneme dále a uvedeme jako příklad hru Age of empires, zjistíme že AI není jen o chování samotných jednotek. V této hře mohl hráč, ovládající určitý národ, hrát proti soupeři, který byl kompletně ovládán počítačem. To znamenalo důstojně simulovat lidského protivníka, budovat město, těžit zdroje, útočit na hráče a bránit se. Tato oblast je velice komplexní a zahrnuje velké množství akcí na různých úrovních. Nepřítel se v této hře řídil určitou množinou pravidel a tak se po jisté době začaly situace opakovat a hra ztrácela na svém kouzle. Jakmile hráč jednou zjistil, jak nepřítele porazit, po jisté době mu to nečinilo větší potíže.

Aby se tedy AI dokázala vyrovnat lidskému hráči, často se v podobných případech používal cheating nebo-li podvádění [1]. To znamená že počítačem řízený protivník nastavený na vyšší obtížnost rychleji těží zdroje, staví budovy a celkově má výhody oproti hráči, které částečně prodlouží dobu, po kterou hráče hra proti počítači baví. Tato technika se ale netěší mezi designery velké oblibě, protože jakmile hráč zjistí, že nepřítel má neprávem výhodu, hra ztrácí na důvěryhodnosti.

Myslím si, že je v poslední době tento problém na ústupu jelikož většina strategických her je orientována na multiplayer hru, tedy hru více hráčů. S masovým rozšířením internetu se tedy vývoj více soustředil tímto směrem a hráči tak soupeří mezi sebou. Tituly jako Warcraft 3 nebo Starcraft 2 jsou velice orientovány na multiplayer hraní a taky velice úspěšné co se prodejností týče.

Jelikož je tato oblast velice rozsáhlá a není možné ji v obsahu této práce obsáhnout z detailního hlediska, uvedeme si jen ty oblasti, které jsou použitelné pro jednoduchou implementaci různých technik použitelných v našem případě. Pokud se na celý problém podíváme z obecného hlediska, tak pro základní funkčnost ekonomických jednotek jsou důležité následující akce. Pohyb, interakce s prostředím, schopnost rozhodnout se o dalším kroku a v případě některých jednotek dodržovat svůj cíl, ke kterému byly vytvořeny. Později si uvedeme příklady použití různých technik nebo jejich kombinací pro různé jednotky, které byly k tomuto účelu implementovány v našem demu.

Hledání nejkratší cesty již bylo popsáno v samostatné kapitole, ale v této kapitole si upřesníme jak se jednotky při pohybu chovají a jakým způsobem je řešena kolize mezi jednotkami samotnými. Může

zde nastat mnoho situací, které je potřeba vyřešit a docílit tak předpokládaného chování.

4.2 Techniky používané v této oblasti

Oblast Game AI je v poslední době jednou z nejrychleji rozvíjejících se. S příchodem rychlých grafických akcelerátorů se nároky na procesor velice snížily a navíc se rozmohly více jádrové procesory, které jsou schopny zpracovávat paralelně několik vláken. V důsledku se tak může více procesorového času použít na zpracování game AI.

Existuje zde velké množství technik použitelných pro implementaci game AI. Od těch jednodušších jako konečné stavové automaty, rozhodovací stromy, vyhledávací metody a algoritmy, až po pokročilejší jako genetické algoritmy, samoučící se rozhodovací stromy a neuronové sítě. Hlavním rozdílem je pohled na tyto techniky jako na celek. Zatímco v akademické sféře se za AI považují velice složité oblasti a techniky, z pohledu herních vývojářů bývá často do této oblasti řazeno více věcí [2]. Vývoj game AI také upřednostňuje různá ad-hoc řešení, je zde kladen důraz na použití heuristiky a také často můžeme vidět velice různé implementace toho samého problému. Samozřejmostí je také používaní různých hacků k dosáhnutí kýženého výsledku. I když to na první pohled nemusí být ihned zřejmé, heuristika hraje významnou roli v mnoha oblastech. Často totiž pro různé oblasti game AI nepotřebujeme dokonalé řešení, ale spíše takové, které se blíží tomu co zamýšlíme. Také se zde klade důraz na rychlost. Celý tento subsystém musí být schopen zpracovávat požadavky pro každý vyrenderovaný frame.

4.2.1 Pohyb

Pohyb je tou nejzákladnější činností, kterou musí jednotky v realtime strategii vykonávat. Celá hra je založena na pohybu jednotek a jejich přesunu z místa na místo a následném provádění různých akcí. Toto platí jak pro ekonomické jednotky, tak také pro vojenské jednotky i pro zvířata, která mohou být poblíž. Uveďme si příklad. Jednotka, která těží dřevo dostane příkaz po svém vytvoření aby začala těžit. Zní to velice jednoduše, ale předchází tomu řada akcí. Dříve než se jednotka vydá na cestu, musí vědět, kde se nachází cíl, v tomto případě strom. Až jednotka bude mít všechny tyto informace, může se vydat na cestu.

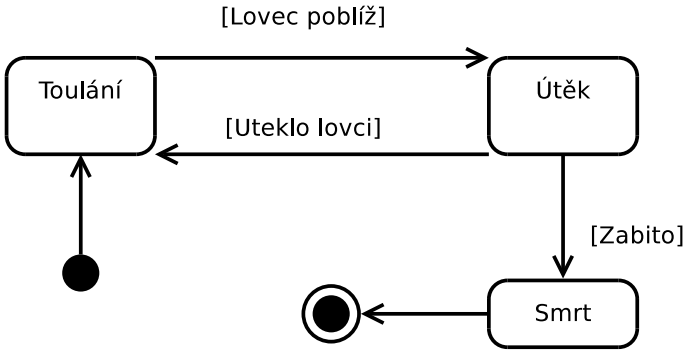
S pohybem je spojeno již zmíněné hledání cesty a vyhýbání se objektům a to statickým i dynamickým. Dále animování jednotky a rotace modelu vzhledem ke směru cesty je dalším problémem, který sice není spojen přímo s AI, ale vzhledem ke své povaze se řeší zároveň s pohybem jednotky. Samotný pohyb je potom podmnožinou různých akcí nebo-li stavů ve kterých se různé akce provádí.

4.2.2 Konečné stavové automaty

Ve stavovém automatu, každá jednotka okupuje právě jeden stav. S každým stavem je spojená akce či více akcí které se provádí po celou dobu co je jednotka v daném stavu. Změna mezi stavy je pak způsobena buď různými podmínkami nebo triggery.

Jak můžeme vidět na obrázku č. 7, takto vypadá jednoduchá stavová logika pro obyčejné zvíře, které se může pohybovat po herní mapě. Zvíře se toulá po herní mapě, ale jakmile zpozoruje lovce, proběhne změna stavu z poklidného toulání na útěk. Pokud zvíře lovci uteče, vrátí se zpět do stavu toulání. Pokud je ovšem zabito, přejde do stavu smrt, což je i stav konečný.

Tato struktura je velice obecná a dá se implementovat mnoha způsoby. Nejjednodušším způsobem implementace je napevno naprogramovaná stavová logika jednotky. To s sebou sice nese výhodu jednoduchosti a rychlé počáteční implementace, nicméně při každé modifikaci musí být kód překompilován. Také pokud komplexita vzroste, může být velice těžké takovouto strukturu dále modifikovat abychom udrželi čitelnost kódu.



Obrázek 7: Jednoduchá ukázka stavového automatu pro zvíře

Pro ukázku si můžeme uvést jednoduchý kód pro obrázek č. 7. Je zde jasně vidět, jak lze ověřit podmínku, která zajišťuje přechod z jednoho stavu do druhého.

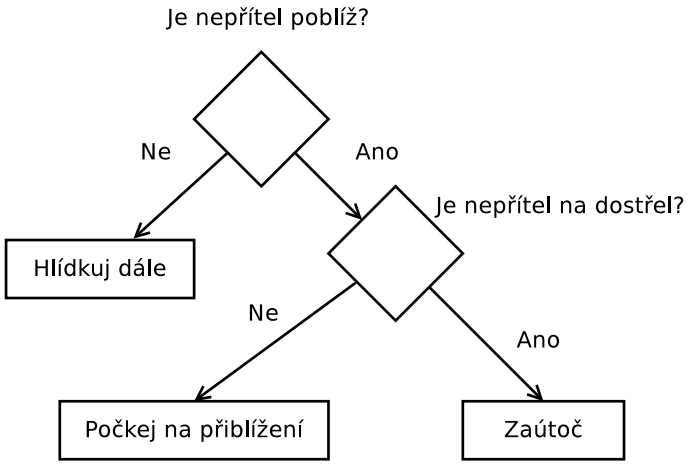
Zdrojový kód 5: Pseudokód pro konečný stavový automat

```
#define WANDER 1
#define FLEE 2
#define DEAD 3

void hardFsm()
{
  if(status == WANDER)
  {
    if(isEnemyNearby())
      status = FLEE;
    else
      wander();
  {
  else if(status == FLEE)
  {
    if(!isEnemyNearby())
      status = WANDER;
    else
      flee();
  }
  else if(status == dead)
    deleteThisUnit();
}
```

4.2.3 Rozhodovací chování a rozhodovací stromy

Důležitým prvkem AI obecně je schopnost provádět rozhodnutí. Jelikož většině akcí předchází rozhodování, je velice výhodné použít strukturu která nám toto umožní. Jednou z takových struktur je rozhodovací strom, neboli Decision tree. Decision tree je jednoduchý, jednoduše implementovatelný a lehce pochopitelný [1]. Je to jedna z nejjednodušších rozhodovacích struktur, která se v AI enginech používá. Velice často se používá pro rozhodování jak na úrovni jednotek samotných tak pro například skupinové rozhodování. Jejich velkou předností je vysoká modularita. I když to dnes není běžné, decision tree může být také učící se.

Každý člen stromu, pokud není listem, představuje určitý problém, nad kterým musí být vyneseno rozhodnutí. Rozhodnutí se provede základě různých okolností. Každé rozhodnutí může být provedeno na základě jak známých faktů, tak i náhodného činitele. Přidáním náhodného činitele docílíme výsledku, který nemusí být předem známý. Listy tohoto stromu pak představují konečné rozhodnutí, které má za výsledek změnu stavu, či provedení nějaké akce.

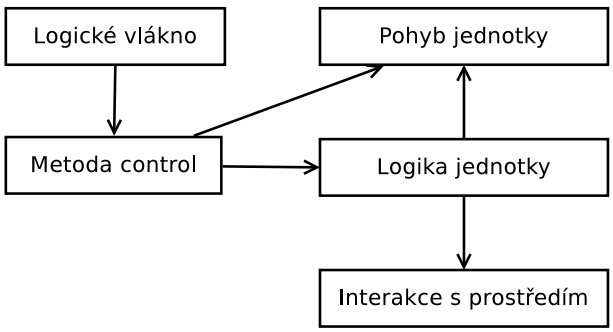


Obrázek 8: Ukázka rozhodovacího stromu

Na obrázku č. 8 vidíme jednoduchý příklad rozhodovacího stromu pro jednotku, která hlídá město. Toto rozhodování nemusí probíhat v každém cyklu. S výhodou se zde dá použít již zmíněné periodické zpracování. To znamená, že jednotka například každé dvě sekundy zhodnotí situaci a provede rozhodnutí.

4.3 Implementace AI logického subsystému

V našem enginu je logický subsystém implementován následovně. Základem všech objektů ve hře je třída Entity. Obsahuje definice metod společné pro všechny ostatní objekty odvozené z této třídy. Pro logický subsystém jsou to hlavně metody **control, goTo, resolveDecision, get2dPosition**. Metoda control je počátečním bodem zpracování logiky uvnitř každé jednotky, jak můžeme vidět na obrázku č. 9



Obrázek 9: Náhled zpracování logické části na úrovni samotné jednotky

Jak jsem později zjistil, důležitou věcí je vhodně rozložit zatížení zpracování celého logického vlákna. Vhodným způsobem je provádět veškeré rozhodovací akce a náročné výpočty z časovou prodlevou a nebo jen pokud je to nutné(Jednotka splnila předchozí úkol). To znamená například veškeré rozhodnutí

provádět periodicky(např. jednou za dvě sekundy). Pokud uvedeme jako příklad jednotku typu Deer, ta provádí zjišťování, zda-li se kolem ní nenachází jiná jednotka typu Hunter. Bylo by velice nevýhodné to zjišťovat v každém cyklu. Podle mého názoru je daleko lepší tyto akce rozdělit tak, aby se rozložila zátěž na logické vlákno rovnoměrně. Nyní je perioda přesně definována. Pokud bychom chtěli jít ještě dále, tak bychom mohli pro každou jednotku periodu lehce pozměnit tak, aby se stejné akce u jednotek vykonávaly v čase jindy. Toho bychom například mohli docílit změnou periody na základě unikátního ID, které má každá entita definované při svém vytvoření a rozdělit tak

Třída DynamicEntity je společným předkem pro všechny objekty schopné pohybu. Přidává virtuální metodu move, která zajišťuje pohyb jednotky v prostoru. Metoda drawPath, která slouží hlavně pro ladící účely vyhledávání cesty, vykresluje aktuální cestu, po které se jednotka bude pohybovat. Pro demonstrační účely jsou momentálně implementovány tyto jednotky.

- Builder - Budovatelská jednotka.
- Gatherer - Jednotka pro těžení zdrojů. Příklad Stavového chování jednotky.
- Hunter - Lovecká jednotka. Slouží pro demonstraci funkce rozhodovacího stromu a interakci s prostředím.
- Taxman - Jednotka vybírající daně.
- Deer - Zvěř. Volně pobíhající po herní mapě. Slouží především jako cíl pro lovce.

Dále si některé jednotky detailněji popíšeme, protože obsahují ukázky implementace různých technik používaných pro demonstrační účely.

4.3.1 Implementace decision tree pro jednotku hunter

Tato jednotka je jednou ze základních ekonomických jednotek ve většině her. Slouží k lovení zvěře, která volně pobíhá po mapě. Jednotka chová následovně. Po vytvoření dojde k inicializaci rozhodovacího stromu. Momentálně je vytvoření rozhodovacího stromu napevno zakódováno do třídy Hunter. Deklarace třídy DecisionTree vypadá následovně.

Zdrojový kód 6: Deklarace třídy DecisionTree

```
class DecisionTree
{
public:
  DecisionTree(DecisionTree *_root, Entity *_entity, bool _final, unsigned int id);
  DecisionTree(Entity *_entity, bool _final, unsigned int id);
  ~DecisionTree(void);

  unsigned int getDecision();
  unsigned int getUnitDecision(unsigned int id);
  void setYes(DecisionTree *_yes);
  void setNo(DecisionTree *_no);
  DecisionTree* getYes();
  DecisionTree* getNo();

private:
  unsigned int id;
  Entity *entity;
```

```
  DecisionTree *root;
  DecisionTree *yes;
  DecisionTree *no;
  bool final;
};
```

Tato obecná definice nám umožňuje použít stejnou strukturu pro různé jednotky. Je nutno dodat, že toto je implementovaný binární rozhodovací strom. Není ale problém upravit kód tak, abychom měli jiné možnosti, než ty, kde se rozlišuje pouze ano/ne a podle toho se pak vybírá potomek. Jak můžeme vidět v deklaraci třídy Hunter, metody buildDecisionTree a resolveDecision slouží k práci s tímto vytvořeným rozhodovacím stromem. Metoda buildDecisionTree obsahuje kód k vytvoření celého stromu pro provádění rozhodnutí. Po vytvoření tohoto stromu je do proměnné dTree v třídě Hunter uložen ukazatel na kořen tohoto stromu. Pokud nyní potřebuje jednotka tohoto typu provést rozhodnutí, zavolá metodu getDecision(dTree.getDecision()), která jednotce vrátí rozhodnutí.

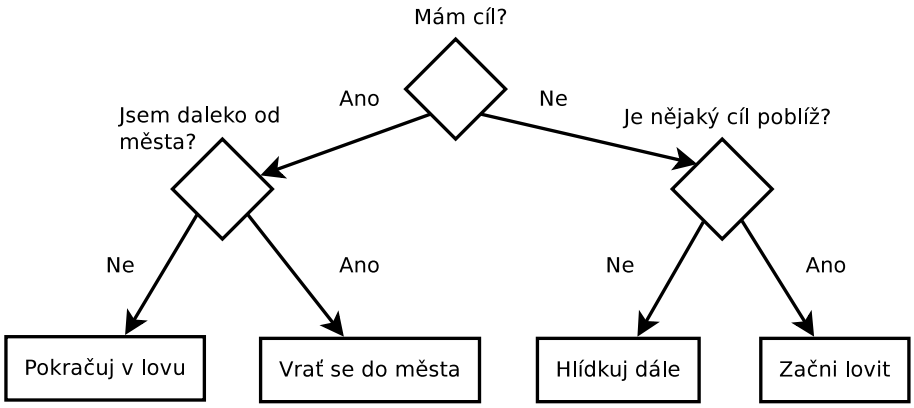
Metoda getDecision rekurzivně projde strom a po dosáhnutí listu, tedy konečného rozhodnutí vrátí jednotce výsledek. To je v nynější implementaci reprezentováno jednoduchým číslem, pro který může být definován stav, do kterého jednotka přejde při rozhodnutí.

Zdrojový kód 7: Deklarace třídy Hunter

```
class Hunter: public Villager
{
public:
  Hunter(glm::vec3 _Pos, float _r, float _speed, std::string _Name,
         std::string _modelFName, ModelPool *_mPool, PathFinder *_pathfinder,
         Terrain::Terrain *_ter);

  ~Hunter(void);
  virtual void AIFunction();
  void buildDecisionTree();
  unsigned int resolveDecision(unsigned int id);
protected:
  Deer *deer;
  DecisionTree *dTree;
  Timer timer2;
};
```

Nyní se tedy jednotka hunter chová následovně. Po jejím vytvoření jednotka ví, že její úkol je lovit zvěř. Takže proběhne prvotní rozhodnutí. Pokud jednotka nemá cíl, v zadaném rozsahu od budovy se bude pokoušet nějaký najít. Jakmile spatří cíl vydá se jej pronásledovat. Pokud mu zvíře uteče, nebo se nachází ve velké vzdálenosti od města, vydá se na cestu zpět. Tato funkčnost má za úkol demonstrovat jednoduché použití rozhodovacích stromů. Jak takovýto strom pro zmíněné chování vypadá můžeme vidět na obrázku č. 10. Abychom ale neprováděli rozhodnutí neustále, je zde zavedena prodleva mezi jednotlivými rozhodnutími. Každá jednotka si vytvoří instanci třídy Timer a podle ní pak v časových úsecích provádí zhodnocení situace a závěrečné zhodnocení.

Pro dosažení lepších výsledků můžeme ve třídě Hunter deklarovat novou proměnnou, např. experience. Při každém ulovení zvířete pak vzroste tato proměnná a na základě ní pak můžeme zvětšit rádius ve kterém se jednotka pohybuje. To nám potom umožní provádět rozhodnutí založené jak na okolním stavu a daných pravidlech, tak vnitřních vlastnostech jednotky, které se časem mění. Pro interakci s okolím používá Hunter třídu Pathfinder pro zjišťování, zda-li je poblíž jednotka typu deer. Pokud ano, znamená to, že je jednotce vrácen ukazatel na danou entity tohoto typu a začne její samotné pronásledování.



Obrázek 10: Rozhodovací strom pro jednotku typu Hunter

4.3.2 Implementace stavového automatu pro jednotku Gatherer a Deer

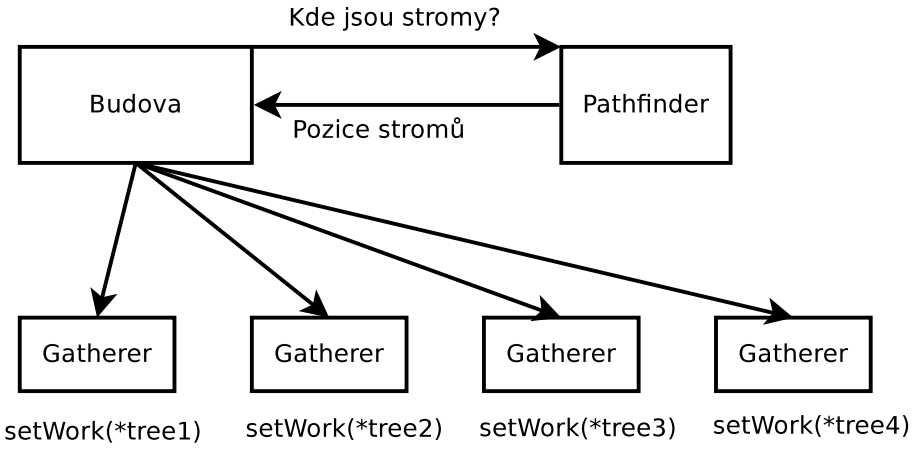
Jednotka gatherer je typickým příkladem ekonomické jednotky. Jejím úkolem je težit jisté zdroje(dřevo, kamení, atd.). Pro implementaci této jednotky, jsem zvolil úplnou autonomnost této jednotky. Nelze ji ovládat hráčem a je centrálně řízena z budovy, která je jejím vlastníkem. Pro ukázku je zde implementována jednotka těžící dřevo. Jednotka obsahuje metodu setWork, kterou je pak ovládána centrální budovou. Po zadání rozkazu již jednotka autonomně provádí svoji práci. Jednotka má nadefinovány stavy, ve kterých se může nalézat. Podle stavu zároveň provádí přidruženou akci. Metoda doWood pak obsluhuje celou akci zpracování jednotlivých stavů.

Zdrojový kód 8: Deklarace třídy Gatherer

```
#define READY 0
#define TO_TREE 1
#define FROM_TREE 2
#define DONE 3
#define NOTHING 4

class Gatherer: public Villager
{
  public:
    Gatherer(glm::vec3 _Pos, float _r, float _speed, std::string _Name,
      std::string _modelFName, ModelPool *_mPool, PathFinder *_pathfinder,
      Terrain::Terrain *_ter);
    ~Gatherer(void);

    virtual void control(float t);
    virtual void goTo(glm::vec2 pos);
    void doWood();
    void setWork(glm::vec2 sTreePosition, glm::vec2 sGatherPosition);
    bool doCut();
    void doSleep();
    void doRun();
    int getStatus();

  protected:
    glm::vec2 TreePosition;
    glm::vec2 GatherPosition;
```

```
    Timer lumberTime;

};
```

Tato jednotka také představuje skupinové řízení. Budova je nadřazena skupině jednotek a vydává rozkazy. Tudíž budova obstará nalezení jednotlivých zdrojů a nastaví jednotkám úkoly. Ty se pak vydají tyto úkoly plnit. Náhled na obrázku č. 11.



Obrázek 11: Centrální řízení ekonomických jednotek budovou

Naproti tomu je jednotka typu Deer naprosto autonomní a dokáže přecházet mezi pevně definovanými stavy, jak můžeme vidět na obrázku č. 7. Jednotka se náhodně pohybuje po herní mapě a simuluje tak zvěř. Abychom chování přiblížili realitě, můžeme ovlivnit vybírání náhodné pozice, kam se jednotka má vydat. První modifikací by bylo vybírání náhodného místa tak, aby se nenacházelo poblíž budov. Tím dosáhneme toho, že se jednotky nebudou pohybovat poblíž měst. Samozřejmě toto nevylučuje možnost, že výsledná náhodná cesta pro tuto jednotku nepovede přes město. Abychom ovlivnili tuto skutečnost, muselo by se zavést různé ohodnocení cest pro různé jednotky a to tak, aby cesta v okolí budov byla pro tuto jednotku dražší a naopak levnější v okolí stromů.

Druhou modifikací by mohlo být pohybování se ve stádech. V konkrétní implementaci by bylo nejjednodušší toho dosáhnout následovně. Každá jednotka tohoto typu by při vytvoření měla proměnnou alpha, která by byla při vytvoření naplněna náhodným číslem. Poté při vyhledávání dalšího bodu pro pohyb by jednotka zjistila, zda-li se v okolí nenachází jednotka s vyšší hodnotou alpha a pokud ano, zvolila by následující bod poblíž této jednotky. Pokud ne, tak by jednoduše pokračovala v náhodném pohybování se. Po čase bychom dosáhli toho, že se jednotky budou pohybovat podle jednotky s nejvyšší hodnotou alpha.

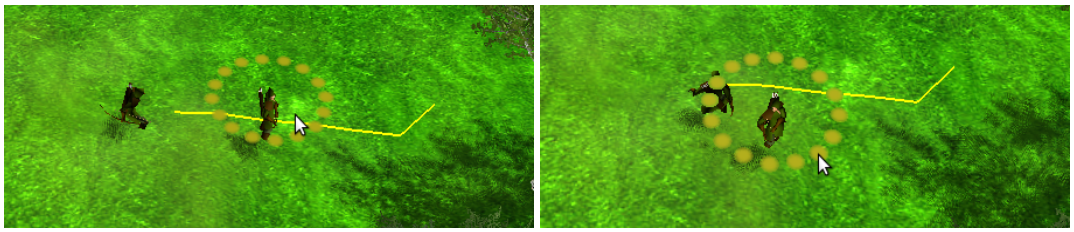
4.4 Implementace AI ve spojení s hledáním cesty

Pathfinding je jeden z nejdůležitějších subsystému logiky. Jelikož je popsán v předchozí kapitole, zde se budu věnovat samotnému chování jednotky při procházení nalezené cesty. Jakmile jednotka dostane cíl, sama požádá pathfinder aby jí vrátil validní cestu. Jednotka pouze předá jako parametry svou aktuální pozici a cílový bod. Jednotka se také chová tak, že pokud je zadán nevalidní bod(Již obsazený, voda), najde si cestu nejblíže k danému bodu. Po navrácení kompletní cesty od pathfinderu se jednotka vydá na cestu. Abychom předešli kolizím, řeší se kolize na úrovni jednotek přímo v pathfinderu. Jednotka pří každém vyjití z jednoho bodu do druhého provede následující sled akcí.

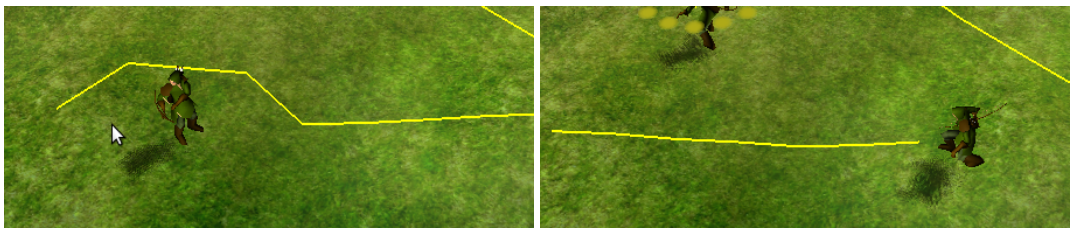
1. Zkontroluje jestli je následující bod volný
2. Pokud ano, zamluví si v pathfinderu následující bod a uvolní stávající
3. Pokud ne, jednotka znova spustí vyhledávání cesty a opakuje celou akci.

Samozřejmě zde existuje velice mnoho situací, které mohou nastat. Jednou z typických situací jsou dvě jednotky, které se pohybují naproti sobě. Momentálně nemůže nastat situace kde dvě jednotky zamlouvají stejný bod, jelikož logika jednotek je zpracována sekvenčně. Tudíž fakticky jednotka která je zpracovávána první si daný bod zamluví. Druhá musí nalézt alternativní cestu.

Další možností jak chování jednotek ve spojením s hledáním cesty přiblížit realitě je implementovat dynamické obnovování cesty. To spočívá v tom, že jednotka v určitých časových úsecích přepočítá cestu. Nemusí však přepočítat celou cestu, neboť to by bylo časově velice náročné. Stačí vybrat úsek který zhruba odpovídá zvolené časové prodlevě mezi jednotlivými obnoveními. Samozřejmě tyto věci jsou velice modifikovatelné a tak je potřeba experimentovat abychom dosáhli chtěného výsledku. Důležitá je perioda obnovování cesty. Pokud bychom uvažovali velice krátký čas mezi obnoveními, mohli bychom tím negativně ovlivnit výkon celého logického subsystému. Proto je důležité zvolit takovou časovou prodlevu



Obrázek 12: Průběh cesty bez obnovování - ke změně cesty dojde až v případě kolize



Obrázek 13: Průběh cesty s obnovováním - cesta se periodicky obnovuje, tudíž pokud překážka uvolní cestu, ta je následně přepočítána

Samotná implementace pohybující entity je následující. Každá jednotka, zvíře a další objekty schopné se pohybovat dědí z DynamicEntity metodu move. Tato metoda je v každém cyklu vykonávána a stará se o logiku pohybu jednotky. Každá pohybující se jednotka má proměnnou std::vector obsahující celou cestu. Cesta jako celek je reprezentována jako body x,y zabalené v datové struktuře glm::vec2 pojmenované moveVector. Jednotka začne vykonávat pohyb, jakmile moveVector obsahuje nějaké data.

Jak můžeme vidět na obrázku č. 12, pokud je obnovování cesty vypnuto, jednotka řeší následující postup až je to nevyhnutelně nutné. V případě zapnutého obnovování cesty, obrázek č. 13, jednotka dopředu zjišťuje, zda-li nemůže dojít ke kolizi a přepočítává si úsek cesty. To samozřejmě platí i pro případ, kdy jednotce uvolní překážka cestu a ona se tak může vydat přímou cestou k cíli. Kdyby zde nebylo obnovování cesty zapnuto, jednotka by vykonala cestu jako kdyby překážka pořád byla na místě.

5 Importování 3D modelů

Abychom mohli demonstrovat veškerou funkčnost popsanou dříve, přirozeně potřebujeme pro všechny entity v herním světě také modely pro jednotky. V 3D realtime strategiích se vetšinou používají modely o nižším počtu polygonů, než je tomu například u FPS her. To je dáno tím, že scéna obsahuje mnoho modelů renderovaných v jednu chvíli, ale také tím, že pohled na herní mapu je oddálen. Existuje veliké množství jak modelovacích nástrojů, tak typů souborů, které tyto modely reprezentují.

Cílem této kapitoly je použití jednoduchých modelů v našem enginu. To zahrnuje načtení daného modelu pomocí loaderu, který následně převede soubor s modelem na data, které můžeme v enginu dále zpracovat. Nebudu se zde zabývat detailním vytvářením modelů a jejich animováním. Spíše budu klást důraz na importovaní a jejich použití v enginu.

Jak již bylo zmíněno, existuje veliké množství modelovacích nástrojů. Jelikož celý engine je postaven na open-source technologiích, tak volba padla na modelovací nástroj Blender. Blender je velice mocný nástroj pro vytváření modelů a jejich animací. Podporuje také export do formátu .obj, což je formát vyvinut firmou Wavefront Technologies. Výhodou toho formátu je, že je dobře čitelný i pro člověka. Obsahuje geometrické informace o daném modelu. Jelikož je to textový soubor, tak pro jeho importování do našeho enginu musíme provést parsování.

5.1 Struktura .obj souboru

Struktura tohoto typu souboru je následující. Soubor obj obsahuje seznam vrcholů a jejich pozici, pozici textur, pozici normálů a seznam bodů z kterých je potom vytvořena plocha.

mtllib	Soubour obsahující materiálové informace	something.mtl
o	Konkrétní objekt v souboru	
v	Geometrické vrcholy:	v x y z
vt	Koordináty pro textury:	vt u v
vn	Normály vrcholů:	vn dx dy dz
f	Plochy	v1/vt1/vn1 v2/vt2/vn2 ... vn/vtn/vnn

Tato struktura může být mnohem složitější, ale pro jednoduchost je zachována tato, která počítá pouze z jedním objektem v obj souboru. Také dovede pracovat pouze s jedinou texturou pro jeden objekt. V budoucnu se počítá z rozšířením o multitexture objekty.

5.2 Importování modelů

Importování modelů je založeno na parsování ASCII souboru .obj do enginu hry. Po načtení souboru se musí jednotlivé složky souboru uložit do struktury, se kterou je možno dále pracovat. Po načtení veškerých údajů o daném objektu je tento model uložen do VBO a dále potom používán pro jednotlivé entity v enginu hry. Modely použité v tomto demu jsou převzaty z volných zdrojů, konkrétně ze stránek `http://thefree3dmodels.com/` a `http://www.blender-models.com/`

5.3 Animace modelů

Jelikož modelové soubory typu obj nepodporují ukládání animace, jediným způsobem je vytvořit model pro každý snímek jednotlivé animace, např. chůze [8]. Tyto modely jsou potom uloženy do tzv. model poolu a postupnou iterací při pohybu se vytváří dojem animovaného pohybu. Každá třída dědící z třídy DynamicEntity má poté implementovány metody, které zajišťují tuto funkčnost.

Zdrojový kód 9: Ukázka kódu pro iteraci mezi modely

```
if (Moving)
{
  mPool->drawModelByID(modelID[(int) animationPos]);
}
else
{
  mPool->drawModelByID(modelID[0]);
}

//-------------------------------------------------------

void dynamicEntity::updateAnimation()
{
  if (Moving)
  {
    animationPos = animationPos + 0.75;

    if (animationPos > modelID.size() - 1)
      animationPos = 1.0;
  }
  else
    animationPos = 1.0;
}
```

Jak můžeme vidět na této ukázce kódu, každá entita schopná pohybu provádí v každém cyklu logického procesu metodu updateAnimation. To zajišťuje již zmíněnou iteraci mezi modely v případě pohybu a vytváří tak animaci pohybu. Pokud se daná entita nehýbe, nastavuje se automaticky ID modelu na první.

Pokud nastane situace, při které jednotka změní rychlost pohybu, musíme upravit rychlost inkrementace proměnné animationPos. Tím dosáhneme věrné animace i při změnách rychlosti a jednotka nebude při pohybu působit komicky.

Poslední věcí, kterou je potřeba při pohybu a jeho animaci vyřešit je správná rotace modelu. Přirozeně požadujeme aby se model natáčel směrem, kterým se jednotka momentálně pohybuje. Toho dosáhneme tak, že v každém cyklu pohybu spočítáme úhel pod kterým se jednotka pohybuje z momentálního bodu do následujícího.

Zdrojový kód 10: Rotace modelu podle cíle pohybu

```
  float c = MoveVector.back().x - Pos.x;
  float b = MoveVector.back().y - Pos.z;
  float rRad = 0.0;

  if (MoveVector.back().y < Pos.z)
  {
    rRad = atan(c / b) + PI;
    r = (rRad * 180.0 / PI);
  }
  else
  {
    rRad = atan(c / b);
    r = rRad * 180.0 / PI;
  }
```

6 Závěr

Cílem této práce bylo seznámit se s architekturou enginu pro RTS hry, konkrétně tedy hlavně s logickým subsystémem a zjistit, které techniky a postupy zde lze uplatnit a výhodně použít tak, abych dosáhl chtěného výsledku.

Jelikož tato práce je součástí většího projektu, který postupně rostl, museli jsme zvážit použití vhodného verzovacího systému a také repositáře, kde projekt budeme uchovávat. Volbou nakonec byl Subversion. Subversion je hojně používaný open-source verzovací systém, částečně inspirován starším Concurrent Version System(CVS). Výhodou SVN je, že díky rozšířenosti existují pluginy pro integraci SVN do různých vývojových prostředí a umožňuje nám tak s repositářem pracovat přímo z něj. To nám umožnilo koordinovat postup při vývoji celého projektu a zároveň velmi usnadnilo udržet si přehled o celkovém stavu aplikace. Projekt jsme uchovávali na Gogole SVN repositářích. Projekt je jako celek postaven výhradně na open-source technologiích a je multiplatformní. Momentálně je spustitelný na systémech Microsoft Windows a Linux.

Podařilo se mi dosáhnout implementace těchto částí logického subsystému do enginu hry a tedy i výsledného dema:

1. Logický subsystém - rozvržení exekuce jednotlivých procesů pro řízení všech objektů, jak ovladatelných tak autonomních, navržení vhodného rozvržení zátěže.

2. Hledání nejkratší cesty - funkční modul, používající A* algoritmus pro hledaní nejkratší cesty. Zahrnuje také zkoumání vlivu různých způsobů odhadování vzdálenosti do zadaného bodu na výslednou cestu a rychlost zpracováni. Tento modul zároveň obstarává různé funkce, které jsou spojeny s průchodností terénu a obsazením jednotlivých pozic.

3. Jednoduchá AI

   (a) Implementace a použití rozhodovacího stromu pro jednotky.

   (b) Implementace a použití stavové logiky pro jednotky.

   (c) Navigace a pohyb v terénu za použití modulu pro hledání cesty.

Tyto jednotlivé části pak tvoří celek, jenž je použit ve výsledném demu, na kterém jsem spolupracoval s Martinem Dorazilem a jehož úkolem bylo vytvořit grafický subsystém za pomocí OpenGL. Jelikož výsledné demo dosáhlo poměrně velké rozsáhlosti, rozhodli jsme se pokračovat ve vývoji a eventuálně rozšířit tým lidí, kteří budou na tomto projektu spolupracovat. V budoucnu bych rád dokončil implementaci jednotného logického subsystému, který pak bude sloužit jako základ pro skriptovanou logiku. Také plánuji vylepšení subsystému pro hledání cesty, to konkrétně zahrnuje implementaci víceúrovňového hledání cesty a vyladění zátěže celého subsystému.

Velice rádi bychom dosavadní postup přetvořili v použitelný RTS engine, na kterém bychom v budoucnu mohli postavit kompletní hru.

Literatura

[1] Millington Ian, John David Funge, *"Artificial intelligence for games"*, Burlington, MA: Morgan Kaufmann/Elsevier, 2009. 303. ISBN 0123747317

[2] SCHWAB, Brian. *"AI game engine programming"* 1. vyd. Boston: Charles River Media, 2004, 594 s. ISBN 15-845-0344-0.

[3] Pittman, Jamey *"Pac-Man Dossier"* [Online] publikováno 16. června 2011. Dostupné z: http://home.comcast.net/ jpittman2/pacman/pacmandossier.html

[4] Xiao Cui, Hao Shi *"A*-based Pathfinding in Modern Computer Games"* [Online] publikováno 1.Ledna 2011. Dostupné z: http://paper.ijcsns.org/07_book/201101/20110119.pdf

[5] Pinter, Marco *"Toward More Realistic Pathfinding"* [Online] publikováno 14.května 2001. Dostupné z: http://www.gamasutra.com/view/feature/3096/toward_more_realistic_pathfinding.php

[6] Victor S.Adamchik *"Binary Heaps"* [Online] publikováno r.2009, dostupné z: http://www.cs.cmu.edu/ adamchik/15-121/lectures/Binary%20Heaps/heaps.html

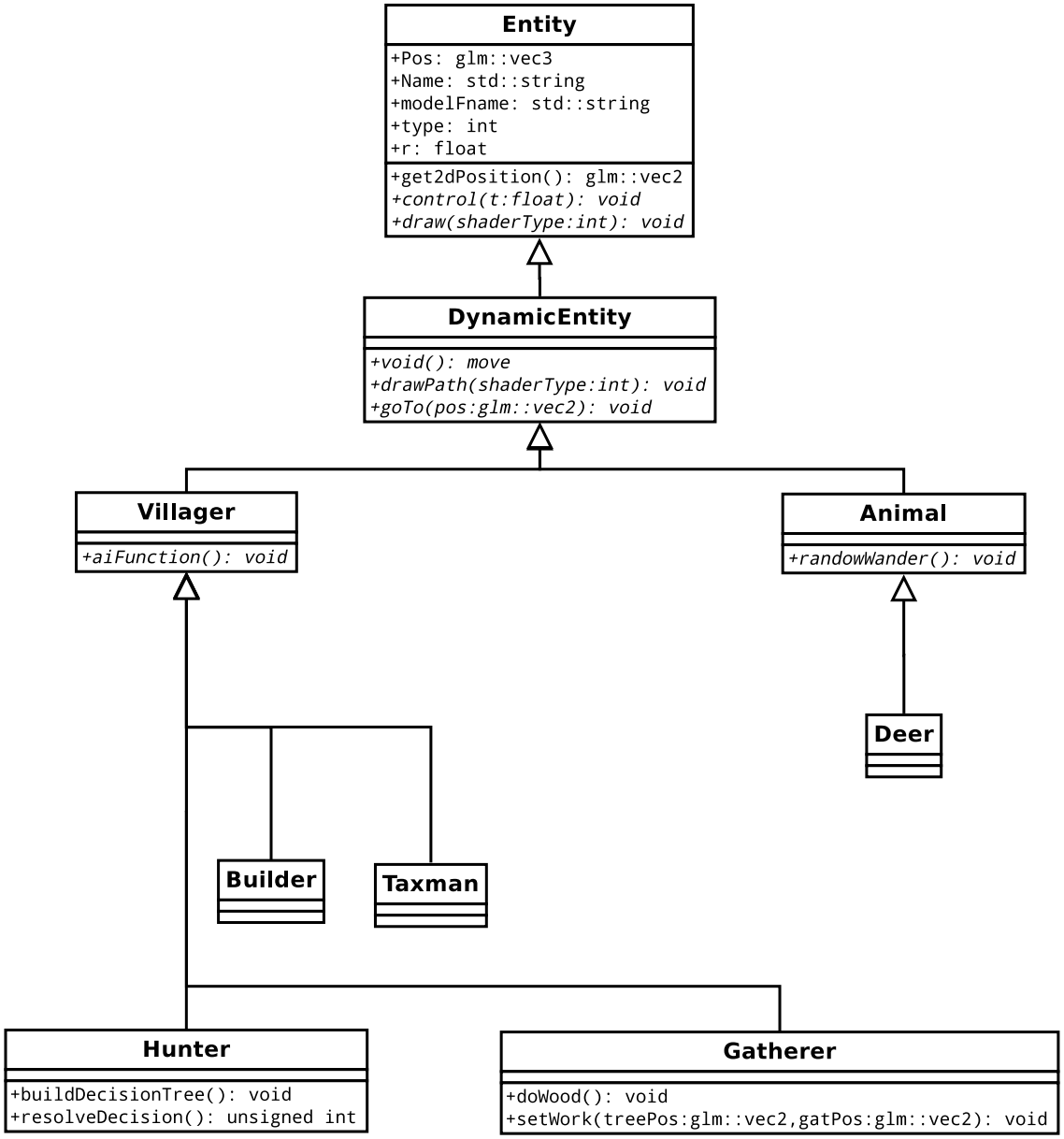
[7] Mikko Laakso, Ari Korhonen, Ville Karavirta *"Priority Queues and Binary Heap"* [Online] publikováno 30.května 2011, dostupné z: http://www.cse.hut.fi/en/research/SVG/TRAKLA2/tutorials/heap_tutorial/taulukkona.html

[8] Bourke, Paul *"Obj file specification"* [Online] dostupné z: http://paulbourke.net/dataformats/obj/

A Screenshot demo aplikace

Obrázek 14: Náhled výsledného enginu. Zapnuto zobrazování cest

B UML Diagram pro třídy odvozené z Entity



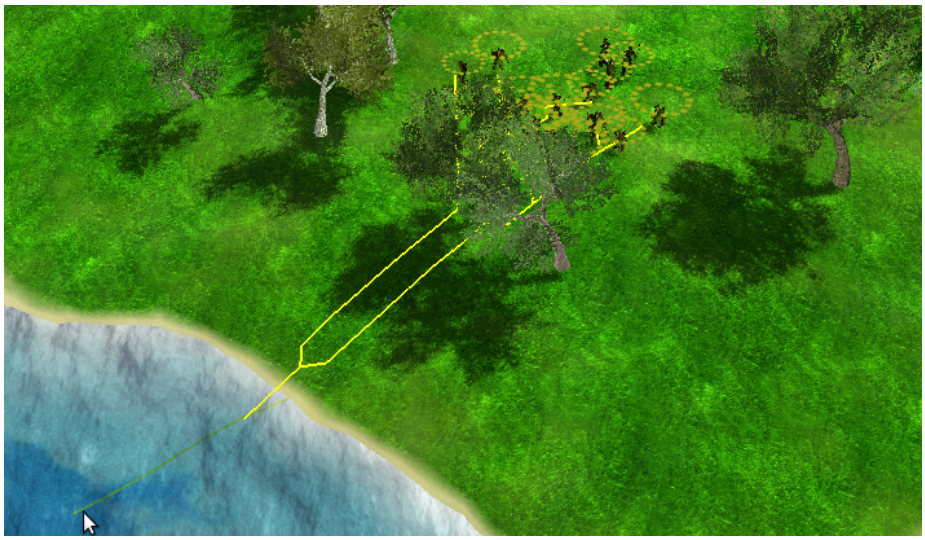
Obrázek 15: Neúplný class diagram pro třídy odvozené z třídy Entity. Vypsány jsou jen metody důležité pro funkční AI

C Definice třídy DecisionTree

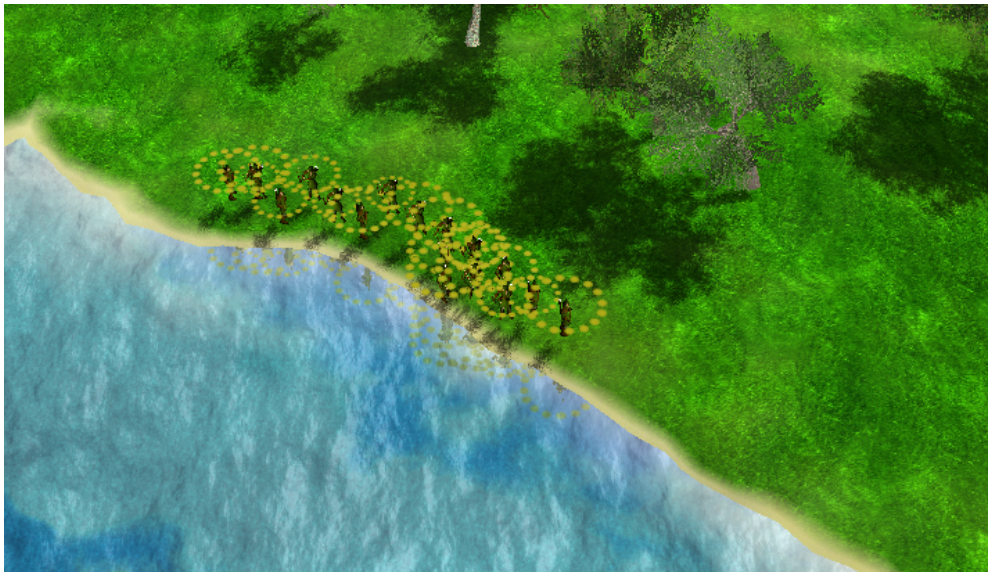
```
#ifdef _WIN32
#include <windows.h>
#endif

#include "DecisionTree.h"

DecisionTree::DecisionTree(DecisionTree *_root, Entity *_entity, bool _final,
    unsigned int _id)
{
  entity = _entity;
  root = _root;
  final = _final;
  id = _id;
  no = NULL;
  yes = NULL;
}

DecisionTree::DecisionTree(Entity *_entity, bool _final, unsigned int _id)
{
  entity = _entity;
  root = NULL;
  final = _final;
  id = _id;
  yes = NULL;
  no = NULL;
}

DecisionTree::~DecisionTree(void){}
void DecisionTree::setYes(DecisionTree *_yes){ yes = _yes; }
void DecisionTree::setNo(DecisionTree *_no){ no = _no; }
DecisionTree* DecisionTree::getYes(){ return yes; }
DecisionTree* DecisionTree::getNo(){ return no;  }

unsigned int DecisionTree::getDecision()
{
  //Get unit decision for specific leaf of tree
  //This method needs to be overriden in each entity using Decision Tree to perform
      evaluation of leaf according to id
  //Method should perform decision based on real facts(known things(speed, range,
      etc..)) & optionaly some random
  unsigned int dec = getUnitDecision(id);

  if (dec == 1 && final != true && yes != NULL)
  {
    //call yes node
    return yes->getDecision();

  }
  else if (dec == 0 && final != true && no != NULL)
  {
    //call no node
    return no->getDecision();
  }
  else if (final)
  {
    //Return final decision. Need's to be defined somewhere else statically
    //Or possibly call method which set units action based on returning id.
```

```
    return id;
  }
  else
  {
    //log error, something has gone wrong. Return 999 in case of failure
    return 999;
  }
}

unsigned int DecisionTree::getUnitDecision(unsigned int id)
{
  return entity->resolveDecision(id);
}
```

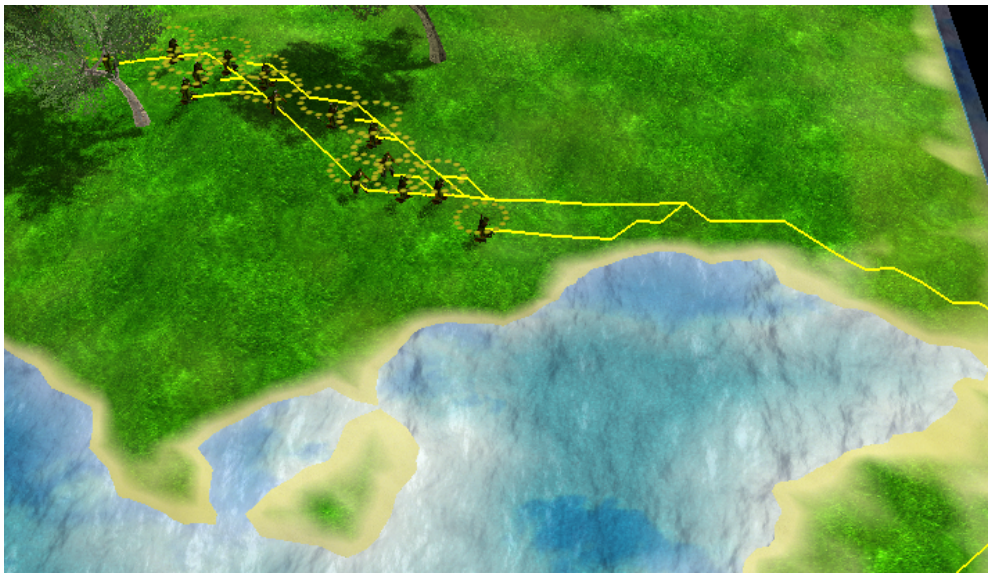
D Ukázka hledání cesty při skupinovém pohybu



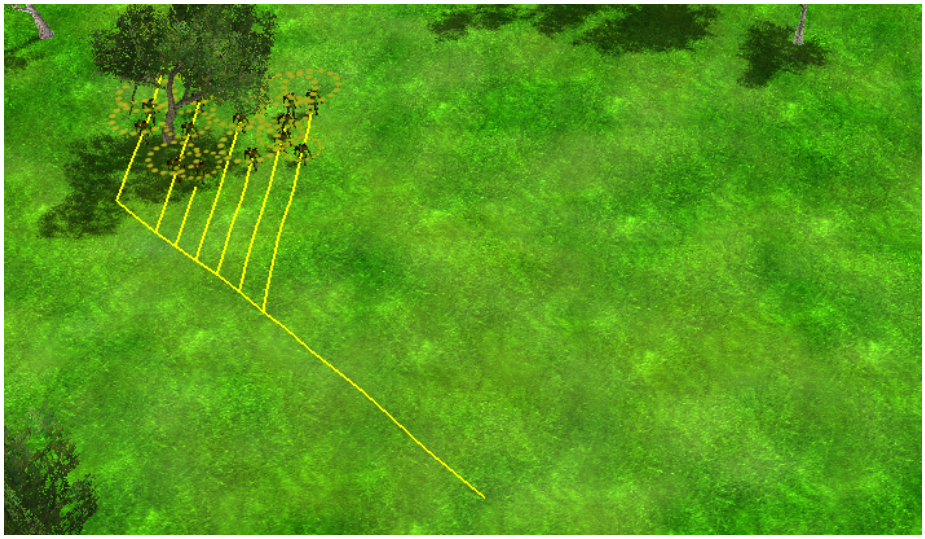
Obrázek 16: Chování jednotek při zadaném cíli ve vodě



Obrázek 17: Takto se jednotky rozestaví v případě nedostupného cíle(voda)



Obrázek 18: Ukázka obcházení překážky při pohybu ve skupině



Obrázek 19: Skupinový pohyb